**Superset ID-6363144**

**Name- Harshita Sinha**

**Engineering Concepts (Design Patterns and Principles)**

**Exercise 1: Implementing the Singleton Pattern**

**Solution:-**

using System;

namespace SingletonPatternExample

{

public class Logger

{

private static readonly Lazy<Logger> \_instance = new Lazy<Logger>(() => new Logger());

private Logger()

{

Console.WriteLine("Logger initialized.");

}

public static Logger Instance

{

get

{

return \_instance.Value;

}

}

public void Log(string message)

{

Console.WriteLine($"[LOG] {DateTime.Now}: {message}");

}

}

class Program

{

static void Main()

{

Console.WriteLine("Starting application...\n");

Logger logger1 = Logger.Instance;

logger1.Log("First log message");

Logger logger2 = Logger.Instance;

logger2.Log("Second log message");

Console.WriteLine($"\nAre logger1 and logger2 the same instance? " +

$"{ReferenceEquals(logger1, logger2)}");

Console.WriteLine("\nPress any key to exit.");

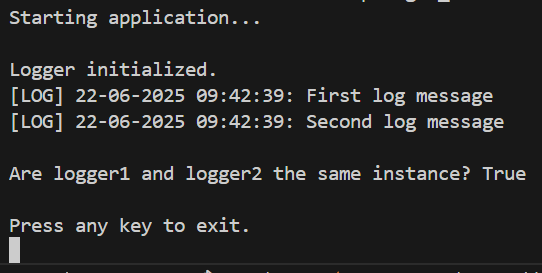
Console.ReadKey();

}

}

}

**Output:-**



**Exercise 2: Implementing the Factory Method Pattern**

**Solution:-**

using System;

namespace FactoryMethodPatternExample

{

    public interface IDocument

    {

        void Open();

    }

    public class WordDocument : IDocument

    {

        public void Open()

        {

            Console.WriteLine("📄 Word Document opened.");

        }

    }

    public class PdfDocument : IDocument

    {

        public void Open()

        {

            Console.WriteLine(" PDF Document opened.");

        }

    }

    public class ExcelDocument : IDocument

    {

        public void Open()

        {

            Console.WriteLine(" Excel Document opened.");

        }

    }

    public abstract class DocumentFactory

    {

        public abstract IDocument CreateDocument();

    }

    public class WordFactory : DocumentFactory

    {

        public override IDocument CreateDocument()

        {

            return new WordDocument();

        }

    }

    public class PdfFactory : DocumentFactory

    {

        public override IDocument CreateDocument()

        {

            return new PdfDocument();

        }

    }

    public class ExcelFactory : DocumentFactory

    {

        public override IDocument CreateDocument()

        {

            return new ExcelDocument();

        }

    }

    class Program

    {

        static void Main()

        {

            Console.WriteLine("🛠️ Document Management System Using Factory Method\n");

            DocumentFactory wordFactory = new WordFactory();

            IDocument word = wordFactory.CreateDocument();

            word.Open();

            DocumentFactory pdfFactory = new PdfFactory();

            IDocument pdf = pdfFactory.CreateDocument();

            pdf.Open();

            DocumentFactory excelFactory = new ExcelFactory();

            IDocument excel = excelFactory.CreateDocument();

            excel.Open();

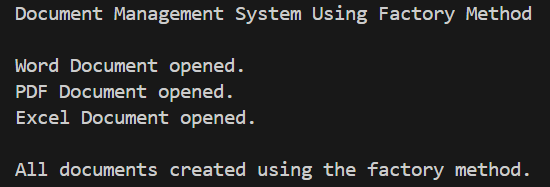
            Console.WriteLine("\n All documents created using the factory method.");

        }

    }

}

**Output:-**



**Exercise 3: Implementing the Builder Pattern**

**Solution:-**

using System;

namespace BuilderPatternExample

{

public class Computer

{

public string CPU { get; private set; }

public string RAM { get; private set; }

public string Storage { get; private set; }

public string GPU { get; private set; }

public string OS { get; private set; }

private Computer(Builder builder)

{

CPU = builder.CPU;

RAM = builder.RAM;

Storage = builder.Storage;

GPU = builder.GPU;

OS = builder.OS;

}

public override string ToString()

{

return $"Computer Configuration:\n" +

$"- CPU: {CPU}\n" +

$"- RAM: {RAM}\n" +

$"- Storage: {Storage}\n" +

$"- GPU: {GPU}\n" +

$"- OS: {OS}";

}

public class Builder

{

public string CPU { get; private set; }

public string RAM { get; private set; }

public string Storage { get; private set; }

public string GPU { get; private set; }

public string OS { get; private set; }

public Builder SetCPU(string cpu)

{

CPU = cpu;

return this;

}

public Builder SetRAM(string ram)

{

RAM = ram;

return this;

}

public Builder SetStorage(string storage)

{

Storage = storage;

return this;

}

public Builder SetGPU(string gpu)

{

GPU = gpu;

return this;

}

public Builder SetOS(string os)

{

OS = os;

return this;

}

public Computer Build()

{

return new Computer(this);

}

}

}

class Program

{

static void Main()

{

Console.WriteLine(" Builder Pattern Example\n");

Computer gamingPC = new Computer.Builder()

.SetCPU("Intel Core i9")

.SetRAM("32GB")

.SetStorage("1TB SSD")

.SetGPU("NVIDIA RTX 4090")

.SetOS("Windows 11")

.Build();

Console.WriteLine(" Gaming PC:\n" + gamingPC);

Console.WriteLine();

Computer officePC = new Computer.Builder()

.SetCPU("Intel Core i5")

.SetRAM("8GB")

.SetStorage("512GB SSD")

.SetOS("Windows 10")

.Build();

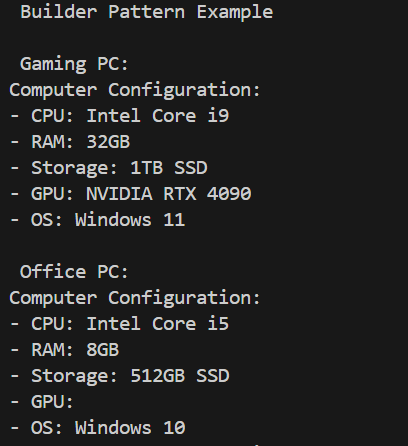
Console.WriteLine(" Office PC:\n" + officePC);

}

}

}

**Output:-**



**Exercise 4: Implementing the Adapter Pattern**

**Solution:-**

using System;

namespace AdapterPatternExample

{

public interface IPaymentProcessor

{

void ProcessPayment(string customerName, double amount);

}

public class PayPalGateway

{

public void MakePayment(string user, double total)

{

Console.WriteLine($"[PayPal] Processing payment of ₹{total} for {user}");

}

}

public class StripeGateway

{

public void SendPayment(double amount, string client)

{

Console.WriteLine($"[Stripe] Payment of ₹{amount} sent for {client}");

}

}

public class RazorpayGateway

{

public void ExecuteTransaction(string name, double money)

{

Console.WriteLine($"[Razorpay] Transaction of ₹{money} completed for {name}");

}

}

public class PayPalAdapter : IPaymentProcessor

{

private PayPalGateway \_paypal = new PayPalGateway();

public void ProcessPayment(string customerName, double amount)

{

\_paypal.MakePayment(customerName, amount);

}

}

public class StripeAdapter : IPaymentProcessor

{

private StripeGateway \_stripe = new StripeGateway();

public void ProcessPayment(string customerName, double amount)

{

\_stripe.SendPayment(amount, customerName);

}

}

public class RazorpayAdapter : IPaymentProcessor

{

private RazorpayGateway \_razorpay = new RazorpayGateway();

public void ProcessPayment(string customerName, double amount)

{

\_razorpay.ExecuteTransaction(customerName, amount);

}

}

class Program

{

static void Main()

{

Console.WriteLine(" Payment Processing System Using Adapter Pattern\n");

IPaymentProcessor paypal = new PayPalAdapter();

IPaymentProcessor stripe = new StripeAdapter();

IPaymentProcessor razorpay = new RazorpayAdapter();

paypal.ProcessPayment("Soumya", 1500.00);

stripe.ProcessPayment("Rupsa", 2200.50);

razorpay.ProcessPayment("Harshita", 999.99);

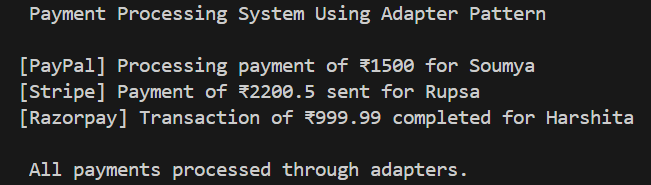
Console.WriteLine("\n All payments processed through adapters.");

}

}

}

**Output:-**



**Exercise 5: Implementing the Decorator Pattern**

**Solution:-**

using System;

namespace DecoratorPatternExample

{

    public interface INotifier

    {

        void Send(string message);

    }

    public class EmailNotifier : INotifier

    {

        public void Send(string message)

        {

            Console.WriteLine($" Email sent: {message}");

        }

    }

    public abstract class NotifierDecorator : INotifier

    {

        protected INotifier \_wrappee;

        public NotifierDecorator(INotifier notifier)

        {

            \_wrappee = notifier;

        }

        public virtual void Send(string message)

        {

            \_wrappee.Send(message);

        }

    }

    public class SMSNotifierDecorator : NotifierDecorator

    {

        public SMSNotifierDecorator(INotifier notifier) : base(notifier) { }

        public override void Send(string message)

        {

            base.Send(message);

            Console.WriteLine($" SMS sent: {message}");

        }

    }

    public class SlackNotifierDecorator : NotifierDecorator

    {

        public SlackNotifierDecorator(INotifier notifier) : base(notifier) { }

        public override void Send(string message)

        {

            base.Send(message);

            Console.WriteLine($" Slack message sent: {message}");

        }

    }

    class Program

    {

        static void Main()

        {

            Console.WriteLine(" Notification System Using Decorator Pattern\n");

            INotifier basicNotifier = new EmailNotifier();

            INotifier smsNotifier = new SMSNotifierDecorator(basicNotifier);

            INotifier fullNotifier = new SlackNotifierDecorator(smsNotifier);

            Console.WriteLine("Sending notification via Email only:");

            basicNotifier.Send("Server is up.");

            Console.WriteLine("\nSending notification via Email + SMS:");

            smsNotifier.Send("Server is down!");

            Console.WriteLine("\nSending notification via Email + SMS + Slack:");

            fullNotifier.Send("Server recovered. All systems go!");

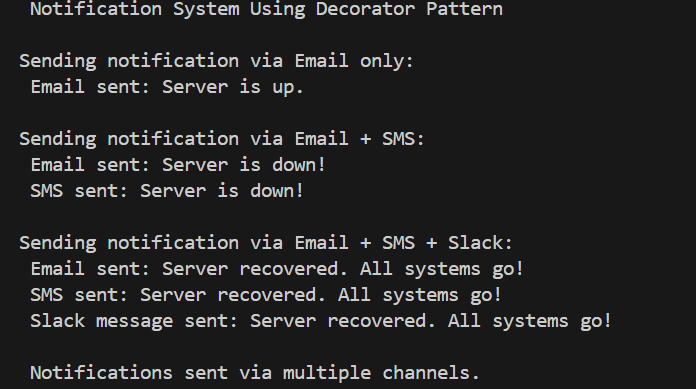
            Console.WriteLine("\n Notifications sent via multiple channels.");

        }

    }

}

**Output:-**



**Exercise 6: Implementing the Proxy Pattern**

**Solution:-**

using System;

namespace ProxyPatternExample

{

    public interface IImage

    {

        void Display();

    }

    public class RealImage : IImage

    {

        private string \_filename;

        public RealImage(string filename)

        {

            \_filename = filename;

            LoadFromDisk();

        }

        private void LoadFromDisk()

        {

            Console.WriteLine($" Loading image '{\_filename}' from remote server...");

        }

        public void Display()

        {

            Console.WriteLine($" Displaying image: {\_filename}");

        }

    }

    public class ProxyImage : IImage

    {

        private RealImage \_realImage;

        private string \_filename;

        public ProxyImage(string filename)

        {

            \_filename = filename;

        }

        public void Display()

        {

            if (\_realImage == null)

            {

                \_realImage = new RealImage(\_filename);

            }

            \_realImage.Display();

        }

    }

    class Program

    {

        static void Main()

        {

            IImage image = new ProxyImage("nature.jpg");

            Console.WriteLine(" First call:");

            image.Display();

            Console.WriteLine("\n Second call:");

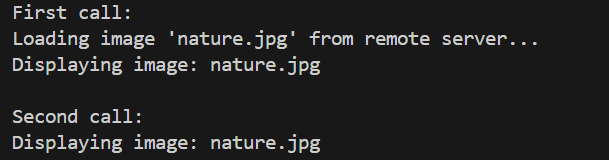
            image.Display();

        }

    }

}

**Output:-**



**Exercise 7: Implementing the Observer Pattern**

**Solution:-**

using System;

using System.Collections.Generic;

namespace ObserverPatternExample

{

    public interface IObserver

    {

        void Update(string stockName, double price);

    }

    public interface IStock

    {

        void Register(IObserver observer);

        void Deregister(IObserver observer);

        void NotifyObservers();

        void SetPrice(double price);

    }

    public class StockMarket : IStock

    {

        private readonly List<IObserver> \_observers = new List<IObserver>();

        private string \_stockName;

        private double \_stockPrice;

        public StockMarket(string stockName)

        {

            \_stockName = stockName;

        }

        public void Register(IObserver observer)

        {

            \_observers.Add(observer);

        }

        public void Deregister(IObserver observer)

        {

            \_observers.Remove(observer);

        }

        public void NotifyObservers()

        {

            foreach (var observer in \_observers)

            {

                observer.Update(\_stockName, \_stockPrice);

            }

        }

        public void SetPrice(double price)

        {

            \_stockPrice = price;

            NotifyObservers();

        }

    }

    public class MobileApp : IObserver

    {

        public void Update(string stockName, double price)

        {

            Console.WriteLine($"📱 Mobile App - {stockName} price updated to ₹{price}");

        }

    }

    public class WebApp : IObserver

    {

        public void Update(string stockName, double price)

        {

            Console.WriteLine($"💻 Web App - {stockName} price updated to ₹{price}");

        }

    }

    class Program

    {

        static void Main()

        {

            StockMarket stock = new StockMarket("TCS");

            IObserver mobile = new MobileApp();

            IObserver web = new WebApp();

            stock.Register(mobile);

            stock.Register(web);

            stock.SetPrice(3580.45);

            Console.WriteLine();

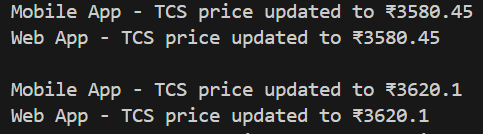
            stock.SetPrice(3620.10);

        }

    }

}

**Output:-**



**Exercise 8: Implementing the Strategy Pattern**

**Solution:-**

using System;

namespace StrategyPatternExample

{

    public interface IPaymentStrategy

    {

        void Pay(double amount);

    }

    public class CreditCardPayment : IPaymentStrategy

    {

        public void Pay(double amount)

        {

            Console.WriteLine($" Paid ₹{amount} using Credit Card.");

        }

    }

    public class PayPalPayment : IPaymentStrategy

    {

        public void Pay(double amount)

        {

            Console.WriteLine($" Paid ₹{amount} using PayPal.");

        }

    }

    public class PaymentContext

    {

        private IPaymentStrategy \_paymentStrategy;

        public PaymentContext(IPaymentStrategy paymentStrategy)

        {

            \_paymentStrategy = paymentStrategy;

        }

        public void SetPaymentStrategy(IPaymentStrategy strategy)

        {

            \_paymentStrategy = strategy;

        }

        public void ExecutePayment(double amount)

        {

            \_paymentStrategy.Pay(amount);

        }

    }

    class Program

    {

        static void Main()

        {

            PaymentContext context = new PaymentContext(new CreditCardPayment());

            context.ExecutePayment(1200);

            context.SetPaymentStrategy(new PayPalPayment());

            context.ExecutePayment(850.50);

        }

    }

}

**Output:-**



**Exercise 9: Implementing the Command Pattern**

**Solution:-**

using System;

namespace CommandPatternExample

{

public interface ICommand

{

void Execute();

}

public class Light

{

public void TurnOn()

{

Console.WriteLine(" Light is ON");

}

public void TurnOff()

{

Console.WriteLine(" Light is OFF");

}

}

public class LightOnCommand : ICommand

{

private Light \_light;

public LightOnCommand(Light light)

{

\_light = light;

}

public void Execute()

{

\_light.TurnOn();

}

}

public class LightOffCommand : ICommand

{

private Light \_light;

public LightOffCommand(Light light)

{

\_light = light;

}

public void Execute()

{

\_light.TurnOff();

}

}

public class RemoteControl

{

private ICommand \_command;

public void SetCommand(ICommand command)

{

\_command = command;

}

public void PressButton()

{

\_command.Execute();

}

}

class Program

{

static void Main()

{

Light livingRoomLight = new Light();

ICommand lightOn = new LightOnCommand(livingRoomLight);

ICommand lightOff = new LightOffCommand(livingRoomLight);

RemoteControl remote = new RemoteControl();

remote.SetCommand(lightOn);

remote.PressButton();

remote.SetCommand(lightOff);

remote.PressButton();

}

}

}

**Output:-**



**Exercise 10: Implementing the MVC Pattern**

**Solution:-**

using System;

namespace MVCPatternExample

{

public class Student

{

public string Name { get; set; }

public string Id { get; set; }

public string Grade { get; set; }

}

public class StudentView

{

public void DisplayStudentDetails(Student student)

{

Console.WriteLine("Student Details:");

Console.WriteLine($"Name: {student.Name}");

Console.WriteLine($"ID: {student.Id}");

Console.WriteLine($"Grade: {student.Grade}");

}

}

public class StudentController

{

private Student \_student;

private StudentView \_view;

public StudentController(Student student, StudentView view)

{

\_student = student;

\_view = view;

}

public void SetStudentName(string name) => \_student.Name = name;

public void SetStudentId(string id) => \_student.Id = id;

public void SetStudentGrade(string grade) => \_student.Grade = grade;

public void UpdateView()

{

\_view.DisplayStudentDetails(\_student);

}

}

class Program

{

static void Main()

{

Student student = new Student();

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.SetStudentName("Harshita Sinha");

controller.SetStudentId("KIIT2026");

controller.SetStudentGrade("A");

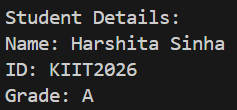
controller.UpdateView();

}

}

}

**Output:-**



**Exercise 11: Implementing Dependency Injection**

**Solution:-**

using System;

namespace DependencyInjectionExample

{

    public interface ICustomerRepository

    {

        string FindCustomerById(int id);

    }

    public class CustomerRepositoryImpl : ICustomerRepository

    {

        public string FindCustomerById(int id)

        {

            return $"Customer #{id} - Harshita Sinha";

        }

    }

    public class CustomerService

    {

        private readonly ICustomerRepository \_repository;

        public CustomerService(ICustomerRepository repository)

        {

            \_repository = repository;

        }

        public void ShowCustomer(int id)

        {

            string customer = \_repository.FindCustomerById(id);

            Console.WriteLine($"Retrieved: {customer}");

        }

    }

    class Program

    {

        static void Main()

        {

            ICustomerRepository repo = new CustomerRepositoryImpl();

            CustomerService service = new CustomerService(repo);

            service.ShowCustomer(101);

        }

    }

}

**Output:-**

